Kapitel 10: Aggregation von Daten und Gruppenoperationen

McKinney, W. (2017). *Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython*. 2. Auflage. Sebastopol, CA [u. a.]: O'Reilly.

Überarbeitet: armin.baenziger@zhaw.ch, 25. Nov. 2020

- Das Kategorisieren eines Datasets und das Anwenden einer Funktion auf jede Gruppe, ob Aggregation oder Transformation, ist häufig eine kritische Komponente eines Datenanalyseworkflows.
- Pandas bietet dazu eine flexible `groupby` -Methode an.
- Hilfreich sind zudem Funktionen für Pivot-Tabellen und Kreuztabellen, welche einen Spezialfall von Pivott-Tabellen darstellen.

In [1]:
```
%autosave 0
```

```
Autosave disabled
```

In [2]:
```
# Wichtige Bibliotheken mit üblichen Abkürzungen laden:
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

In [3]:
```
# Damit Plots direkt im Notebook erscheinen:
%matplotlib inline
```

GroupBy-Mechanismen

- Gruppenoperationen lassen sich mit dem Begriff **"Split-Apply-Combine"** beschreiben.
- In der ersten Phase des Prozesses werden Daten, die in einem Pandas-Objekt enthalten sind basierend auf einem oder mehreren Schlüsseln in Gruppen aufgeteilt.
- Das Teilen wird auf einer bestimmten Achse eines Objekts ausgeführt. Zum Beispiel kann ein DataFrame in seinen Zeilen (Achse = 0) oder seinen Spalten (Achse = 1) gruppiert werden.
- Sobald dies erledigt ist, wird eine Funktion auf jede Gruppe angewendet, wodurch ein neuer Wert erzeugt wird.
- Abschliessend werden die Ergebnisse all dieser Funktionsanwendungen zu einem Ergebnisobjekt zusammengefasst.
- Die folgende Abbildung aus dem Lehrmittel stellt den GroupBy-Mechanismus dar:

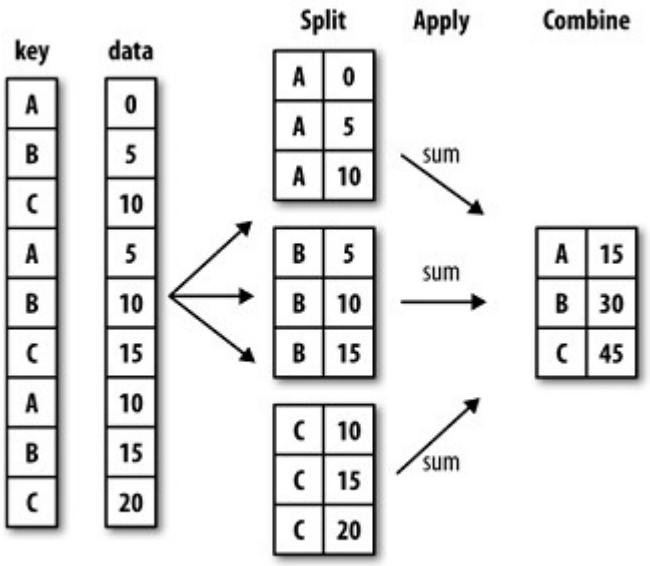


Figure 10-1. Illustration of a group aggregation

Erstes Beispiel:

In [4]:
```
df = pd.DataFrame({'key'  : list('ABCABCABC'),
                   'data' : [0, 2, 5, 3, 1, 7, 2, 0, 4]})
df
```

Out[4]:

	key	data
0	A	0
1	B	2
2	C	5
3	A	3
4	B	1
5	C	7
6	A	2
7	B	0
8	C	4

In [5]:
```
grouped = df.data.groupby(df.key)
grouped   # Dieses Objekt ist ein sog. GroupBy-Objekt
```

Out[5]:
```
<pandas.core.groupby.generic.SeriesGroupBy object at 0x000001642F2
5D048>
```

Das GroupBy-Objekt kann nun verwendet werden, um beispielsweise Gruppenstatistiken zu erstellen. Im Folgenden summieren wir die Werte in `data` für jede Gruppe in `key` separat auf.

In [6]:
```
grouped.sum()
```

Out[6]:
```
key
A     5
B     3
C    16
Name: data, dtype: int64
```

Weiteres Beispiel:

In [7]:
```
np.random.seed(12345)
df = pd.DataFrame({'key1' : ['a', 'a', 'b', 'b', 'a'],
                   'key2' : ['one', 'two', 'one', 'two', 'one'],
                   'data1' : np.random.randint(low=1, high=7, size=
5),
                   'data2' : np.random.randint(low=1, high=7, size=
5)})
df
```

Out[7]:

	key1	key2	data1	data2
0	a	one	3	2
1	a	two	6	6
2	b	one	6	3
3	b	two	2	6
4	a	one	5	2

In [8]:
```
df.data1.groupby(df.key1).mean()
# data1 nach key1 gruppieren und Mittelwerte berechnen
```

Out[8]:
```
key1
a    4.666667
b    4.000000
Name: data1, dtype: float64
```

In [9]:
```
means = df.data1.groupby([df.key1, df.key2]).mean()
# Gruppierung nach key1 und danach key2.
means # Es entsteht eine Series mit hierarchischem Index
```

Out[9]:
```
key1  key2
a     one     4
      two     6
b     one     6
      two     2
Name: data1, dtype: int32
```

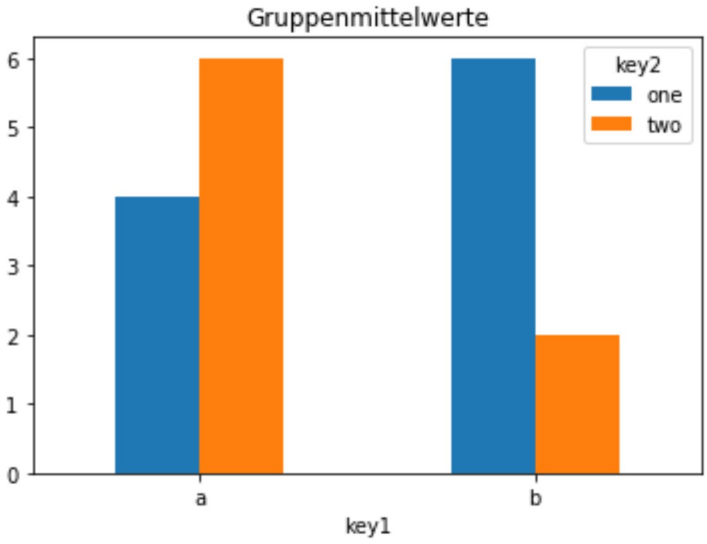
Zur Erinnerung: Mit `unstack` können wir eine (zweite per Default) Hierarchieebene in die Spalten drehen.

In [10]:
```
means.unstack()
```

Out[10]:

key2	one	two
key1		
a	4	6
b	6	2

In [11]:
```
means.unstack().plot.bar(
    title='Gruppenmittelwerte', rot=0);
```



Weiteres Beispiel: In obigen Beispielen sind die Gruppenschlüssel Series bzw. Spalten von DataFrames. Die Gruppenschlüssel können aber *irgendwelche Arrays sein, solange sie die richtige Länge haben!*

In [12]:
```
df # Beispieldatei nochmals betrachten
```

Out[12]:

	key1	key2	data1	data2
0	a	one	3	2
1	a	two	6	6
2	b	one	6	3
3	b	two	2	6
4	a	one	5	2

In [13]:
```
Gruppen = np.array(['G1', 'G2', 'G2', 'G1', 'G1'])
df.groupby(Gruppen).mean()
```

Out[13]:

	data1	data2
G1	3.333333	3.333333
G2	6.000000	4.500000

Hinweis: Die Mittelwerte für `key1` und `key2` fehlen oben, da die Merkmale nicht numerisch sind.

Häufig befinden sich die Gruppierungsinformationen im selben DataFrame wie die Daten, die analysiert werden sollen. In diesem Fall können Spaltennamen (ob Zeichenfolgen, Zahlen oder andere Python-Objekte) als Gruppenschlüssel übergeben werden:

In [14]:
```
df.groupby('key1').mean()
```

Out[14]:

key1	data1	data2
a	4.666667	3.333333
b	4.000000	4.500000

In [15]:
```
df.groupby(['key1', 'key2']).mean()
```

Out[15]:

key1	key2	data1	data2
a	**one**	4	2
	two	6	6
b	**one**	6	3
	two	2	6

Ungeachtet der Zielsetzung bei der Verwendung von `groupby` ist `size` eine allgemein nützliche GroupBy-Methode, die eine *Series* mit den Gruppengrössen zurückgibt:

In [16]:
```
df.groupby(['key1', 'key2']).size()
```

Out[16]:
```
key1  key2
a     one     2
      two     1
b     one     1
      two     1
dtype: int64
```

Im Gegensatz zu `size` werden bei `count` die Anzahl Werte (ohne Fehlwerte!) *pro Spalte* ausgegeben.

In [17]:
```
df.groupby(['key1', 'key2']).count()
```

Out[17]:

		data1	data2
key1	**key2**		
a	**one**	2	2
	two	1	1
b	**one**	1	1
	two	1	1

Da keine `NaN` existieren, haben wir in beiden Spalten die gleichen Werte.

Zwei Wege zu gruppieren

In [18]:
```
df.groupby('key1').data1.median()
```

Out[18]:
```
key1
a    5
b    4
Name: data1, dtype: int32
```

In [19]:
```
df.data1.groupby(df.key1).median()
# Achtung, df.key1 hier, nicht 'key1', da
# key1 nicht mehr in df.data1 enthalten ist!
```

Out[19]:
```
key1
a    5
b    4
Name: data1, dtype: int32
```

- Beide Zeilen führen zum gleichen Resultat.
- Die erste Zeile ist prägnanter.
- *Die zweite Zeile ist insb. in grossen Datensätzen vorzuziehen, da weiger Daten aggregiert werden müssen.*

Kontrollfragen:

In [20]:
```
# Gegeben:
dflohn = pd.read_pickle('../weitere_Daten/dflohn.pkl')
dflohn_sample = (dflohn.sample(5, random_state=13)
                 .sort_values(['Geschlecht', 'Zivilstand']))
dflohn_sample
```

Out[20]:

Person	Lohn	Geschlecht	Alter	Zivilstand
15	8681.0	m	23	v
38	5333.0	m	62	vw
84	9502.0	m	20	vw
44	6945.0	w	54	l
63	4888.0	w	19	v

In [21]:
```
# Frage 1: Was ist der Output?
dflohn_sample.groupby('Geschlecht').Lohn.min()
```

Out[21]:
```
Geschlecht
m    5333.0
w    4888.0
Name: Lohn, dtype: float64
```

In [22]:
```
# Frage 2: Was ist der Output?
dflohn_sample.groupby(
    ['Geschlecht', 'Zivilstand']).Alter.min()
```

Out[22]:
```
Geschlecht  Zivilstand
m           v             23
            vw            20
w           l             54
            v             19
Name: Alter, dtype: int32
```

In [23]:
```
# Frage 3: Was ist der Output?
Abteilung = [1, 2, 2, 1, 1]
dflohn_sample.Lohn.groupby(Abteilung).max()
```

Out[23]:
```
1    8681.0
2    9502.0
Name: Lohn, dtype: float64
```

Aggregation von Daten

- **Aggregationen nennt man Datenumwandlungen, die skalare Werte aus Arrays erzeugen.**
- Die vorhergehenden Beispiele haben mehrere von ihnen verwendet.
- Viele häufig verwendete Aggregationen haben optimierte `groupby` -Implementierungen. Es sind dies: `count` , `sum` , `mean` , `median` , `std/var` , `min/max` , `prod` , `first/last` (erster und letzter Nicht- `NaN` -Wert).
- Zudem kann jede Methode aufgerufen werden, die auch für das gruppierte Objekt definiert ist. Beispiel:

In [24]:
```
# Übersichtliche Darstellung des Beispiel-DataFrames:
df2 = df.drop('key2', axis=1).sort_values(['key1', 'data1'])
df2
```

Out[24]:

	key1	data1	data2
0	a	3	2
4	a	5	2
1	a	6	6
3	b	2	6
2	b	6	3

In [25]:
```
gruppiert = df2.groupby('key1')
gruppiert.quantile(0.25) # 25%-Quantil = 1. Quartil
# Hinweis: Bei so wenigen Werten ist die Bestimmung
# des 1. Quartils nicht wirklich sinnvoll.
```

Out[25]:

	data1	data2
key1		
a	4.0	2.00
b	3.0	3.75

Es fuktionieren auch Methoden wie `describe`, obwohl sie streng genommen keine Aggregationen sind:

In [26]:
```
gruppiert['data1'].describe()
```

Out[26]:

	count	mean	std	min	25%	50%	75%	max
key1								
a	3.0	4.666667	1.527525	3.0	4.0	5.0	5.5	6.0
b	2.0	4.000000	2.828427	2.0	3.0	4.0	5.0	6.0

- Schliesslich ist es auch möglich, *eigene* Aggregationen zu verwenden.
- Um eigene Aggregationsfunktionen zu verwenden, übergibt man eine Funktion, die ein Array aggregiert, an die **`aggregate`** - oder **`agg`** -Methode:

In [27]:
```
def spannweite(x):
    return x.max() - x.min()
gruppiert.agg(spannweite)
```

Out[27]:

	data1	data2
key1		
a	3	4
b	4	3

In [28]:
```
# Exkurs: oder direkt mit einer Lambda-Funktion
# (keine explizite Funktionsdefinition nötig):
gruppiert.agg(lambda x: x.max() - x.min())
```

Out[28]:

key1	data1	data2
a	3	4
b	4	3

Weitere Funktionalitäten dargestellt am "Trinkgelddatensatz":

In [29]:
```
tips = pd.read_csv('../examples/tips.csv')
# Variable hinzufügen, welche Trinkgeld als
# Prozent des Rechnungstotals ausdrückt:
tips['tip_pct'] = tips.tip / tips.total_bill
tips.head()
```

Out[29]:

	total_bill	tip	smoker	day	time	size	tip_pct
0	16.99	1.01	No	Sun	Dinner	2	0.059447
1	10.34	1.66	No	Sun	Dinner	3	0.160542
2	21.01	3.50	No	Sun	Dinner	3	0.166587
3	23.68	3.31	No	Sun	Dinner	2	0.139780
4	24.59	3.61	No	Sun	Dinner	4	0.146808

In [30]:
```
grouped = tips.groupby(['day', 'smoker'])
grouped.mean()    # Mittelwerte
```

Out[30]:

		total_bill	tip	size	tip_pct
day	smoker				
Fri	No	18.420000	2.812500	2.250000	0.151650
	Yes	16.813333	2.714000	2.066667	0.174783
Sat	No	19.661778	3.102889	2.555556	0.158048
	Yes	21.276667	2.875476	2.476190	0.147906
Sun	No	20.506667	3.167895	2.929825	0.160113
	Yes	24.120000	3.516842	2.578947	0.187250
Thur	No	17.113111	2.673778	2.488889	0.160298
	Yes	19.190588	3.030000	2.352941	0.163863

In [31]:
```
# Berechnungen nur für eine Variable:
grouped['tip_pct'].mean()
```

Out[31]:
```
day   smoker
Fri   No        0.151650
      Yes       0.174783
Sat   No        0.158048
      Yes       0.147906
Sun   No        0.160113
      Yes       0.187250
Thur  No        0.160298
      Yes       0.163863
Name: tip_pct, dtype: float64
```

In [32]:
```
grouped['tip_pct'].agg('mean')  # gleiches Ergebnis
```

Out[32]:
```
day   smoker
Fri   No        0.151650
      Yes       0.174783
Sat   No        0.158048
      Yes       0.147906
Sun   No        0.160113
      Yes       0.187250
Thur  No        0.160298
      Yes       0.163863
Name: tip_pct, dtype: float64
```

In [33]:
```
# Mit agg können wir auch mehrere Aggregationen
# gleichzeitig durchführen:
grouped['tip_pct'].agg(['mean', 'std', spannweite])
# 'mean' und 'std' sind Abkürzungen für np.mean
# und np.std.
```

Out[33]:

		mean	std	spannweite
day	**smoker**			
Fri	**No**	0.151650	0.028123	0.067349
	Yes	0.174783	0.051293	0.159925
Sat	**No**	0.158048	0.039767	0.235193
	Yes	0.147906	0.061375	0.290095
Sun	**No**	0.160113	0.042347	0.193226
	Yes	0.187250	0.154134	0.644685
Thur	**No**	0.160298	0.038774	0.193350
	Yes	0.163863	0.039389	0.151240

Kontrollfrage:

In [34]:
```
# Gegeben:
dflohn_sample.sort_values('Zivilstand')
```

Out[34]:

Person	Lohn	Geschlecht	Alter	Zivilstand
44	6945.0	w	54	l
15	8681.0	m	23	v
63	4888.0	w	19	v
38	5333.0	m	62	vw
84	9502.0	m	20	vw

In [35]:
```
# Frage: Was ist der Output?
dflohn_sample.groupby('Zivilstand'
                      ).Alter.agg(['size', 'sum'])
```

Out[35]:

Zivilstand	size	sum
l	1	54
v	2	42
vw	2	82

Die Apply-Methode

Die allgemeinste GroupBy-Methode ist `apply`, welche Gegenstand der folgenden Ausführungen ist. Mit `apply` können Funktionen entlang einer Achse ausgeführt werden.

In [36]:
```
dflohn.groupby('Geschlecht').Lohn.apply(np.mean)
```

Out[36]:
```
Geschlecht
m    5838.918367
w    5850.380000
Name: Lohn, dtype: float64
```

Wie zuvor gezeigt, könnte man diese Aggregation auch mit `agg` oder direkt mit `mean` umsetzen.

In [37]:
```
dflohn.groupby('Geschlecht').Lohn.agg(np.mean)
```

Out[37]:
```
Geschlecht
m    5838.918367
w    5850.380000
Name: Lohn, dtype: float64
```

In [38]:
```
# oder:
dflohn.groupby('Geschlecht').Lohn.agg('mean')
```

Out[38]:
```
Geschlecht
m    5838.918367
w    5850.380000
Name: Lohn, dtype: float64
```

In [39]:
```
# oder direkt:
dflohn.groupby('Geschlecht').Lohn.mean()
```

Out[39]:
```
Geschlecht
m    5838.918367
w    5850.380000
Name: Lohn, dtype: float64
```

Mit `apply` können aber neben Aggregationen weitere Funktionen auf Gruppen angewendet werden. Kehren wir hierzu zum vorherigen Trinkgeld-Datensatz zurück. Angenommen wir wollen eine Funktion schreiben, welche die `n` (Default `n=3`) grössten Werte der Spalte `by` (Default `by=tip_pct`) zurückgibt.

In [40]:
```
def top(df, n=3, by='tip_pct'):
    # "Positional-Argument" (df) vor "Keyword-Argumenten (n, by)"
    return df.sort_values(by=by, ascending=False)[:n]

top(tips, by='tip', n=1)  # höchstes (absolutes) Trinkgeld
```

Out[40]:

	total_bill	tip	smoker	day	time	size	tip_pct
170	50.81	10.0	Yes	Sat	Dinner	3	0.196812

Ohne weitere Argumente gelten die Defaults, also `n=3` und `by=tip_pct` , also die drei höchsten *prozentualen* Trinkgelder relativ zum Rechnungsbetrag.

In [41]:
```
top(tips)
```

Out[41]:

	total_bill	tip	smoker	day	time	size	tip_pct
172	7.25	5.15	Yes	Sun	Dinner	2	0.710345
178	9.60	4.00	Yes	Sun	Dinner	2	0.416667
67	3.07	1.00	Yes	Sat	Dinner	1	0.325733

Wenn wir nun beispielsweise nach der Spalte `smoker` gruppieren und `apply` mit dieser Funktion aufrufen, erhalten wir Folgendes:

In [42]:
```
tips.groupby('smoker').apply(top)
```

Out[42]:

		total_bill	tip	smoker	day	time	size	tip_pct
smoker								
No	**232**	11.61	3.39	No	Sat	Dinner	2	0.291990
	149	7.51	2.00	No	Thur	Lunch	2	0.266312
	51	10.29	2.60	No	Sun	Dinner	2	0.252672
Yes	**172**	7.25	5.15	Yes	Sun	Dinner	2	0.710345
	178	9.60	4.00	Yes	Sun	Dinner	2	0.416667
	67	3.07	1.00	Yes	Sat	Dinner	1	0.325733

Wir erhalten somit die höchsten drei prozentualen Trinkgleder *pro Gruppe*. Auch hier könnte man die Defaults überschreiben:

In [43]:
```
# Pro Gruppe die zwei höchsten absoluten Trinkgelder:
tips.groupby('smoker').apply(top, by='tip', n=2)
```

Out[43]:

		total_bill	tip	smoker	day	time	size	tip_pct
smoker								
No	**212**	48.33	9.00	No	Sat	Dinner	4	0.186220
	23	39.42	7.58	No	Sat	Dinner	4	0.192288
Yes	**170**	50.81	10.00	Yes	Sat	Dinner	3	0.196812
	183	23.17	6.50	Yes	Sun	Dinner	4	0.280535

Wir erhalten nun die zwei höchsten (absoluten) Trinkgelder pro Gruppe (Nichtraucher, Raucher).

Kontrollfrage:

In [44]:
```
# Gegeben:
dflohn_sample
```

Out[44]:

	Lohn	Geschlecht	Alter	Zivilstand
Person				
15	8681.0	m	23	v
38	5333.0	m	62	vw
84	9502.0	m	20	vw
44	6945.0	w	54	l
63	4888.0	w	19	v

In [45]:
```
# Frage: Was ist der Output?
dflohn_sample.groupby('Geschlecht').apply(
             top, by='Lohn', n=1)
# Wir erhalten den höchsten Lohn nach Geschlecht.
```

Out[45]:

		Lohn	Geschlecht	Alter	Zivilstand
Geschlecht	**Person**				
m	**84**	9502.0	m	20	vw
w	**44**	6945.0	w	54	l

Beispiel: Standardabweichung der Tagesrenditen separat pro Jahr

Als Beispiel betrachten wir einen Finanzdatensatz, der von Yahoo-Finance stammt, mit Tagesendkursen einiger Aktien und dem S&P500-Index (Symbol SPX):

In [46]:
```
close_px = pd.read_csv('../examples/stock_px_2.csv',
                       parse_dates=True, index_col=0)
close_px.head()  # die ersten 5 Zeilen
```

Out[46]:

	AAPL	MSFT	XOM	SPX
2003-01-02	7.40	21.11	29.22	909.03
2003-01-03	7.45	21.14	29.24	908.59
2003-01-06	7.45	21.52	29.96	929.01
2003-01-07	7.43	21.93	28.95	922.93
2003-01-08	7.28	21.31	28.83	909.93

In [47]:
```
close_px.tail()   # die letzten 5 Zeilen
```

Out[47]:

	AAPL	MSFT	XOM	SPX
2011-10-10	388.81	26.94	76.28	1194.89
2011-10-11	400.29	27.00	76.27	1195.54
2011-10-12	402.19	26.96	77.16	1207.25
2011-10-13	408.43	27.18	76.37	1203.66
2011-10-14	422.00	27.27	78.11	1224.58

Die praktische Methode `pct_change` berechnet die prozentaulen Veränderungen aus den Kursdaten (also die Renditen).

In [48]:
```
returns = close_px.pct_change().dropna()
# Wir verwenden dropna, da am Anfang des DataFrames
# durch die Renditeberechnungen NaN entstanden.
returns.head()
```

Out[48]:

	AAPL	MSFT	XOM	SPX
2003-01-03	0.006757	0.001421	0.000684	-0.000484
2003-01-06	0.000000	0.017975	0.024624	0.022474
2003-01-07	-0.002685	0.019052	-0.033712	-0.006545
2003-01-08	-0.020188	-0.028272	-0.004145	-0.014086
2003-01-09	0.008242	0.029094	0.021159	0.019386

Standardabweichungen der Tagesrenditen über den ganzen Zeitraum:

In [49]:
```
returns.std()
```

Out[49]:
```
AAPL    0.024486
MSFT    0.017721
XOM     0.016713
SPX     0.013472
dtype: float64
```

Als nächstes berechnen wir die Korrelationen der Aktien mit dem Aktienindex (SPX) *pro Jahr*. Zuerst halten wir fest, dass man die Jahre wie folgt aus dem Datum ziehen kann (Details folgen im Kapitel 11):

In [50]:
```
returns.index.year
```

Out[50]:
```
Int64Index([2003, 2003, 2003, 2003, 2003, 2003, 2003, 2003, 2003,
2003,
            ...
            2011, 2011, 2011, 2011, 2011, 2011, 2011, 2011, 2011,
2011],
           dtype='int64', length=2213)
```

Wir können somit `returns.index.year` als Gruppierungsvektor übergeben:

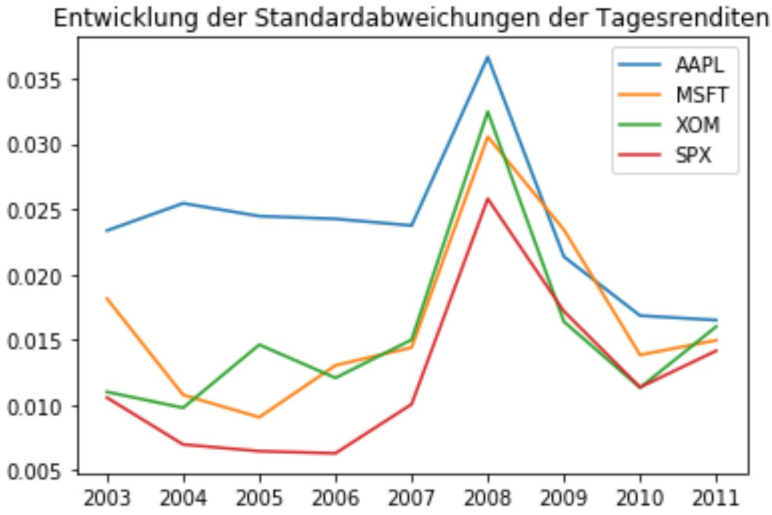
In [51]:
```
Std_pro_Jahr = returns.groupby(returns.index.year).std()
Std_pro_Jahr
```

Out[51]:

	AAPL	MSFT	XOM	SPX
2003	0.023368	0.018156	0.011016	0.010578
2004	0.025457	0.010785	0.009797	0.006988
2005	0.024474	0.009073	0.014629	0.006478
2006	0.024265	0.013049	0.012089	0.006315
2007	0.023757	0.014411	0.015012	0.010070
2008	0.036666	0.030562	0.032472	0.025811
2009	0.021369	0.023429	0.016393	0.017188
2010	0.016856	0.013847	0.011339	0.011372
2011	0.016510	0.014970	0.016019	0.014163

Die Standardabweichungen steigen mit der Finanzkrise an und sinken dann wieder. Wir können dies auch graphisch verdeutlichen.

In [52]:
```
Std_pro_Jahr.plot(
    title='Entwicklung der Standardabweichungen der Tagesrenditen
');
```



Kontrollfrage:

In [53]:
```
# Gegeben:
Auto = pd.read_csv('../weitere_Daten/Auto.csv', sep=';')
Auto.origin.replace({1: 'USA', 2: 'Europa', 3:'Japan'}, inplace=Tru
e)
Auto.tail()
```

Out[53]:

	mpg	cylinders	displacement	horsepower	weight	acceleration	year	origin	nar
387	27.0	4	140.0	86	2790	15.6	82	USA	fc musta
388	44.0	4	97.0	52	2130	24.6	82	Europa	pick
389	32.0	4	135.0	84	2295	11.6	82	USA	dod rampa
390	28.0	4	120.0	79	2625	18.6	82	USA	fc rang
391	31.0	4	119.0	82	2720	19.4	82	USA	che s-

In [54]:
```
# Gegeben:
def MittlereAnzahlBuchstaben(Zeichenkette):
    return Zeichenkette.str.len().mean()
```

In [55]:
```
# Frage: Was bedeutet der Output?
Auto.groupby('origin')['name'].apply(MittlereAnzahlBuchstaben)
# Anzahl Buchstaben der Autonamen nach Herkunft.
# In den USA sind die Namen am längsten.
```

Out[55]:
```
origin
Europa    13.882353
Japan     14.139241
USA       17.383673
Name: name, dtype: float64
```

Pivot-Tabellen und Kreuztabellierung

In [56]:
```
tips.head()
```

Out[56]:

	total_bill	tip	smoker	day	time	size	tip_pct
0	16.99	1.01	No	Sun	Dinner	2	0.059447
1	10.34	1.66	No	Sun	Dinner	3	0.160542
2	21.01	3.50	No	Sun	Dinner	3	0.166587
3	23.68	3.31	No	Sun	Dinner	2	0.139780
4	24.59	3.61	No	Sun	Dinner	4	0.146808

Angenommen wir wollen Gruppenmittelwerte arrangiert nach `day` und `smoker`:

In [57]:
```
tips.groupby(['day', 'smoker']).mean()
```

Out[57]:

		total_bill	tip	size	tip_pct
day	**smoker**				
Fri	**No**	18.420000	2.812500	2.250000	0.151650
	Yes	16.813333	2.714000	2.066667	0.174783
Sat	**No**	19.661778	3.102889	2.555556	0.158048
	Yes	21.276667	2.875476	2.476190	0.147906
Sun	**No**	20.506667	3.167895	2.929825	0.160113
	Yes	24.120000	3.516842	2.578947	0.187250
Thur	**No**	17.113111	2.673778	2.488889	0.160298
	Yes	19.190588	3.030000	2.352941	0.163863

Diese Tabelle ist der Default der Methode `pivot_table` . Somit hätten wir die Tabelle auch wie folgt erhalten (die Spalten folgen aber einer anderen Reihenfolge):

In [58]:
```
tips.pivot_table(index=['day', 'smoker'])
```

Out[58]:

		size	tip	tip_pct	total_bill
day	**smoker**				
Fri	**No**	2.250000	2.812500	0.151650	18.420000
	Yes	2.066667	2.714000	0.174783	16.813333
Sat	**No**	2.555556	3.102889	0.158048	19.661778
	Yes	2.476190	2.875476	0.147906	21.276667
Sun	**No**	2.929825	3.167895	0.160113	20.506667
	Yes	2.578947	3.516842	0.187250	24.120000
Thur	**No**	2.488889	2.673778	0.160298	17.113111
	Yes	2.352941	3.030000	0.163863	19.190588

Die Methode erlaubt mehr: Es sollen die Grössen der Restaurantbesuchergruppen nach Tageszeit, Wochentag und ob sie rauchen oder nicht untersucht werden.

In [59]:
```
tips.pivot_table(
    values  = 'size',    # values: Welche Variable soll ausgewertet werden (Default Mittelwert)
    index   = ['time', 'day'], # index: Gruppierung/Aufgliederung im Index
    columns = 'smoker',        # columns: Aufgliederung in den Spalten
    aggfunc = 'mean')          # Die Aggregationsfunktion 'mean' ist der Default
                               # und könnte somit weggelassen werden.
```

Out[59]:

	smoker	No	Yes
time	day		
Dinner	Fri	2.000000	2.222222
	Sat	2.555556	2.476190
	Sun	2.929825	2.578947
	Thur	2.000000	NaN
Lunch	Fri	3.000000	1.833333
	Thur	2.500000	2.352941

Inklusive "Randstatistiken" mit dem Argument `margins=True` :

In [60]:
```
tips.pivot_table(values  = ['size'],
                 index   = ['time', 'day'],
                 columns = 'smoker',
                 margins=True)
```

Out[60]:

		size		
	smoker	No	Yes	All
time	day			
Dinner	Fri	2.000000	2.222222	2.166667
	Sat	2.555556	2.476190	2.517241
	Sun	2.929825	2.578947	2.842105
	Thur	2.000000	NaN	2.000000
Lunch	Fri	3.000000	1.833333	2.000000
	Thur	2.500000	2.352941	2.459016
All		2.668874	2.408602	2.569672

Um eine andere Aggregationsfunktion zu verwenden (als `mean`), übergibt man diese an `aggfunc` . Beispielsweise erhalten wir mit `len` oder `'count'` (`'count'` , nicht `count`) die absoluten Häufigkeiten.

In [61]:
```
tab = tips.pivot_table(values  = ['total_bill'],
                       index   = ['time', 'smoker'],
                       columns = 'day',
                       aggfunc='count',
                       margins=True)
tab
```

Out[61]:

		total_bill				
	day	Fri	Sat	Sun	Thur	All
time	smoker					
Dinner	No	3.0	45.0	57.0	1.0	106
	Yes	9.0	42.0	19.0	NaN	70
Lunch	No	1.0	NaN	NaN	44.0	45
	Yes	6.0	NaN	NaN	17.0	23
All		19.0	87.0	76.0	62.0	244

Wenn einige Zellen leer bzw. `NaN` sind, kann es sinnvoll sein, einen `fill_value` zu übergeben:

In [62]:
```
tab = tips.pivot_table(values='size',
                       index=['time', 'smoker'],
                       columns='day',
                       aggfunc='count', fill_value=0,
                       margins=True)
tab
```

Out[62]:

	day	Fri	Sat	Sun	Thur	All
time	smoker					
Dinner	No	3	45	57	1	106
	Yes	9	42	19	0	70
Lunch	No	1	0	0	44	45
	Yes	6	0	0	17	23
All		19	87	76	62	244

Kontrollfrage:

In [63]:
```
# Gegeben:
Auto.head()
```

Out[63]:

	mpg	cylinders	displacement	horsepower	weight	acceleration	year	origin	name
0	18.0	8	307.0	130	3504	12.0	70	USA	chevrolet chevelle malibu
1	15.0	8	350.0	165	3693	11.5	70	USA	buick skylark 320
2	18.0	8	318.0	150	3436	11.0	70	USA	plymouth satellite
3	16.0	8	304.0	150	3433	12.0	70	USA	amc rebel sst
4	17.0	8	302.0	140	3449	10.5	70	USA	ford torino

In [64]:
```
# Frage: Wie ist der Output zu interpretieren?
subset = Auto.cylinders.isin([4, 6, 8])  # nur Autos mit 4, 6 oder 8 Zylindern
Auto[subset].pivot_table(values  = 'mpg',
                         index   = 'origin',
                         columns = 'cylinders',
                         margins=True)
```

Out[64]:

cylinders	4	6	8	All
origin				
Europa	28.106557	20.100000	NaN	27.613846
Japan	31.595652	23.883333	NaN	30.978667
USA	28.013043	19.645205	14.963107	20.033469
All	29.283920	19.973494	14.963107	23.445455

- Die Werte in der Tabelle entsprechen den durchschnittlichen Meilen pro Gallone Kraftstoff.
- Autos mit weniger Zylinder sind kraftstoffeffizienter (nicht unbedingt kausal, da Autos mit mehr Zylinder typischerweise grösser bzw. schwerer sind).
- Japanische Autos sind kraftstoffeffizienter als europäische und diese wiederum effizienter als US-Autos.
- Acht Zylinder haben nur US-Autos im Datensatz.

Kreuztabellen

Eine **Kreuztabelle** (auch **Kontingenztabelle** oder **Kontingenztafel** genannt) ist ein *Spezialfall einer Pivot-Tabelle*, die Gruppen**häufigkeiten** darstellt. Beispiele:

In [65]:
```
pd.crosstab(index=Auto.cylinders,
            columns=Auto.origin,
            margins=True)
```

Out[65]:

origin	Europa	Japan	USA	All
cylinders				
3	0	4	0	4
4	61	69	69	199
5	3	0	0	3
6	4	6	73	83
8	0	0	103	103
All	68	79	245	392

Lesebeispiel: Es gibt im Datensatz 199 Autos mit 4 Zylindern, wobei 61 davon aus Europa stammen.

Das nächste Beispiel wiederholt eine Tabelle, die wir zuvor mit `pivot_table` erstellt haben.

In [66]:
```
pd.crosstab(index   = [tips.time, tips.smoker],
            columns = tips.day,
            margins = True)
```

Out[66]:

day		Fri	Sat	Sun	Thur	All
time	smoker					
Dinner	**No**	3	45	57	1	106
	Yes	9	42	19	0	70
Lunch	**No**	1	0	0	44	45
	Yes	6	0	0	17	23
All		19	87	76	62	244

Weiteres Beispiel:

In [67]:
```
dflohn = pd.read_pickle('../weitere_Daten/dflohn.pkl')
dflohn.head()
```

Out[67]:

	Lohn	Geschlecht	Alter	Zivilstand
Person				
1	4107.0	m	40	g
2	5454.0	m	47	vw
3	3719.0	m	41	g
4	6194.0	m	18	v
5	NaN	m	27	v

In [68]:
```
pd.crosstab(index = dflohn.Geschlecht,
            columns = dflohn.Zivilstand,
            margins = True)
```

Out[68]:

Zivilstand	g	l	v	vw	All
Geschlecht					
m	11	10	13	16	50
w	14	10	14	12	50
All	25	20	27	28	100

11 Personen im Datensatz sind männlich (m) und geschieden (g) usw.

In [69]:
```
# Relative Häufigkeitstabelle:
pd.crosstab(index     = dflohn.Geschlecht,
            columns   = dflohn.Zivilstand,
            normalize = True,
            margins   = True)
```

Out[69]:

Zivilstand	g	l	v	vw	All
Geschlecht					
m	0.11	0.1	0.13	0.16	0.5
w	0.14	0.1	0.14	0.12	0.5
All	0.25	0.2	0.27	0.28	1.0

11% der Presonen im Datensatz sind männlich (m) und geschieden (g) usw.

Bedingte Häufigkeitstabellen:

Oft ist es hilfreich, wenn man bedingte relative Häufigkeiten ausweist um Strukturunterschiede in den Daten festzustellen.

In [70]:
```
tab1 = pd.crosstab(
           index     = dflohn.Geschlecht,
           columns   = dflohn.Zivilstand,
           normalize = 'columns',     # Spalten auf 1 normieren.
           margins   = True)
tab1.round(2)
```

Out[70]:

Zivilstand	g	l	v	vw	All
Geschlecht					
m	0.44	0.5	0.48	0.57	0.5
w	0.56	0.5	0.52	0.43	0.5

Die relativen Häufigkeiten sind pro Spalte (Zivilstand) "normalisiert". Somit sind die Spaltentotale jeweils 1 (100%). Beispielsweise sind (im Datensatz) unter den Geschiedenen (g) 44% Männer und 56% Frauen. Insgesamt (All) sind im Datensatz 50% Männer und 50% Frauen.

Kontrollfrage

In [71]:
```
# Frage: Wie ist der erste Wert in der Tabelle (unter "m" und "g") zu interpretieren?
tab2 = pd.crosstab(index     = dflohn.Geschlecht,
                   columns   = dflohn.Zivilstand,
                   normalize = 'index',
                   margins   = True)
tab2.round(2)
```

Out[71]:

Zivilstand	g	l	v	vw
Geschlecht				
m	0.22	0.2	0.26	0.32
w	0.28	0.2	0.28	0.24
All	0.25	0.2	0.27	0.28

Antwort: 22% der Männer (!) sind geschieden.

Fazit

- Die Beherrschung der Datengruppierungstools von Pandas ist sowohl bei der Datenbereinigung als auch bei der statistischen Analyse oder Modellierung sehr hilfreich.
- Im nächsten Kapitel befassen wir uns mit Zeitreihendaten.